Flexible semi-automatic support for type migration of primitive types for C/C++ programs

Richárd Szalay [1], Zoltán Porkoláb [1]

[1] Department of Programming Languages & Compilers, Eötvös Loránd University, Budapest, Hungary
szalayrichard@inf.elte.hu, gsd@ik.elte.hu

Introduction

Type systems are crucial tools in the hands of developers. The compilers' type checking is the first line of defence against programmer error. Unfortunately, developers often not use type systems to their capacity. This is most prevalent with trivial values expressed by fundamental (e.g. `int`) or library (e.g. `string`) types. Developers often resort to embedding the meaning of a variable's expressed value in its name [1] and not its type. Perhaps the most widely-known example of such weak interfaces is the *Mars Climate Orbiter* incident [2]. Due to a design misunderstanding, some program modules used different units of measurement conceptually while only communicating in terms of "numbers", which resulted in the incident. Compilers can catch such mistakes, but only if the program uses types more elaborate than `double`.

Type migration is the process of changing the types of program elements. Conventionally, one would design the new types in advance, specify a mapping from old types to the new ones, and perform the migration. If some operations are left undefined, the code does not compile, making the code incomprehensible to developer tools.

Instead, we propose an approach which combines code comprehension and type migration into the same process. By utilising the type system of an existing language but in an orthogonal plane, we are able to interactively walk the developer through the discovery of how a newly introduced type should look like.

Fictive Types

Our approach uses the **fictive types** ("*ft*") annotation technique to embed additional information about program elements into the source code. The highlight of annotations is that compiler and tool vendors are allowed to specify their own set. Tools are encouraged to ignore – maybe with an accompanying warning – annotations they do not understand. The following example shows a local variable whose "real" type is `int` and *fictive* type temperature. Existing compilers and tools interact with the real type only, and the project can be continually released, while tools more aware interact with the fictive type. Fictive types express only the "set membership" relation – "SensorTemp *is-a* temperature".

```
[[ft(temperature)]] int SensorTemp;
```

I(n)tera(c)tive refactoring process

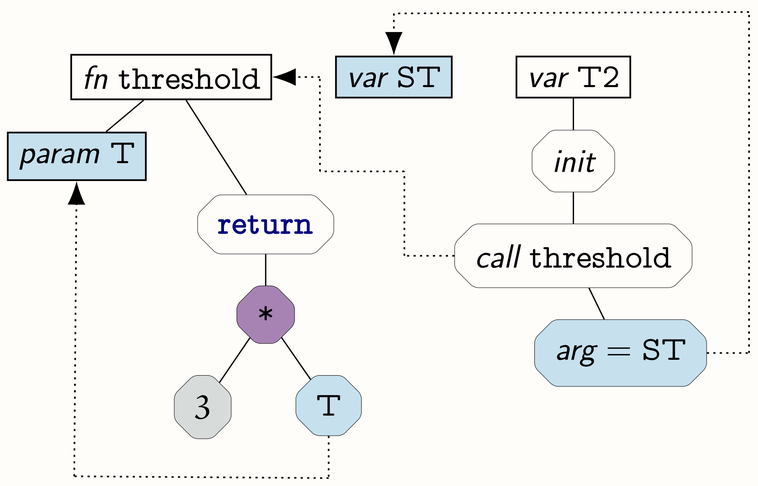
The refactoring consists of three steps. The propagations step are executed in a saturating fix-point iteration, where the developer is asked on-demand to provide additional input.

Taint seeding

Consider the previous example wherein the developer decided some variable *is-a* "temperature". This variable is passed to a function somewhere, and that function returns triple the value, and this is emitted to some output.

```
int threshold(int T) { return 3 * T; }
int T2 = threshold(SensorTemp);
write(T2);
```

Code analysis and transformation is executed on the *abstract syntax tree* (*AST*). A simplified AST for the previous example – with the type colouring – is shown below.



Propagation

The propagation is executed via a modified compiler built upon the LLVM Compiler Infrastructure's *Clang* project. It is a monotonic operation where more program elements receive a taint. In **Round 1**, the propagation tool discovers that the fictively typed variable is passed to the function parameter – a trivial propagation via assignment. This turns `threshold` into a function taking "`temperature`". As each round is as expansive as possible, the type expression `<?> * temperature` is investigated, where a *recoverable error* is generated for the undefined operation.

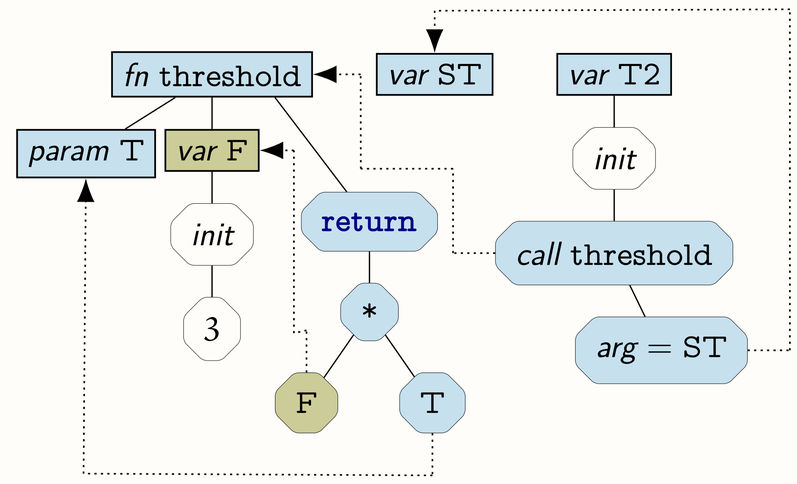
```
test.cpp:1:32: warning: use of undefined fictive
    operator '<?> * temperature'
    3 * T;
    ~ ^
test.cpp:1:30: note: left operand is 'int' literal,
    configure overload or refactor into variable
```

Errors are recoverable by the developer changing the parameters of the environment, which in practice is done via an interactive configuration file. If the developer, for example, decides to define `3` to be of fictive type factor, and that `factor * temperature` is temperature, the code is transformed, and the next round begins.

```
int threshold([[ft(temperature)]] int T) {
    [[ft(factor)]] int F = 3;
    return F * T;
}
int T2 = threshold(SensorTemp);
write(T2);
```

Round 2 associates the result of the `*` operation with the return value of the function, and the assignment of the function's result to the local variable results in tainting the local variable. The user at this point could decide that `write` is a library function which they do not wish to change the type of. This results in an explicit type cast away from the new type. As there are no more production rules to take, the algorithm terminates successfully.

```
[[ft(temperature)]] int
threshold([[ft(temperature)]] int T) {
    [[ft(factor)]] int F = 3;
    return F * T;
}
[[ft(temperature)]] int T2 =
    threshold(SensorTemp);
write(T2);
```



If an *irrecoverable error* is discovered, the algorithm terminates with the error. For example, a function containing two `return` statements that return values of different fictive types would classify as an irrecoverable error.

Transition to strong type

After the successful propagation, the tool generates *strong types* for the fictive types used and rewrites the code to use the new types. Developers are then encouraged to add invariants, additional semantics, or apply existing refactoring operations on these types. The rewritten variables explicitly wrap and unwrap at interface boundaries [3].

```
class temperature { /* ... */ };
class factor      { /* ... */ };
temperature operator *(factor, temperature);
temperature threshold(temperature T) {
    factor F{3}; // Explicit cast from int literal.
    return F * T;
}
temperature SensorTemp = /* ... */;
temperature T2 = threshold(SensorTemp);
print(static_cast<int>(T2));
```

The project is now enhanced with increased type safety. Importantly, future development efforts that would violate the interface discovered for this type will result in an error message from any standard-compliant compiler.

```
some_function(T2 + 1.5);

test.cpp:11:18: error: invalid operands to binary
    '+' expression ('temperature' and 'double')
    some_function(T2 + 1.5);
                  ~~ ^ ~~~
```

References

A full-length *SANER* paper in the proceedings with the same title accompanies this poster.

[1] A. Rice, E. Aftandilian, C. Jaspan, E. Johnston, M. Pradel, and Y. Arroyo-Paredes, "Detecting argument selection defects," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 104:1–104:22, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133928

[2] A. G. Stephenson, L. S. LaPiana, D. R. Mulville, P. J. Rutledge, F. H. Bauer, D. Folta, G. A. Dukeman, R. Sackheim, and P. Norvig, "Mars Climate Orbiter – mishap investigation report," National Aeronautics and Space Administration (NASA), Tech. Rep., 11 1999. [Online]. Available: http://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf

[3] T. Winters, "Non-atomic refactoring and software sustainability," in *Proceedings of the 2nd International Workshop on API Usage and Evolution*, ser. WAPI '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2–5. [Online]. Available: http://dl.acm.org/doi/10.1145/3194793.3194794

International Conference on Software Analysis, Evolution, and Reengineering (SANER), March 15 – 18, 2022, Honolulu, Hawaii, USA